IN THE SPECIFICATION:

Please replace paragraphs [0003] and [0004] with the following amended paragraphs:

[0003] In a typical application, a request (e.g., a user request) is received by an action router (event handler). Based on the data included with the user request the action router determines which function of the application (the "application action function") will handle the user request. In some cases the application action function calls a user interface renderer to display the appropriate output to the user. Throughout the processing of the request the application maintains state information. State information is the information that the application builds and uses during request processing. Such information includes application variables and their values for tracking who the user is, what they are doing and what they can do next during subsequent processing. For example, state information may include (but is not limited to) a user identification (ID) ~~ID~~ and password that the user authenticates with, the visual representation of a current user interface being presented to the user (e.g., whether the interface is terse or verbose, the absence or presence of a taskbar, and other preferences), contents of a shopping cart, and the history of user requests.

[0004] The foregoing generally describes the processing performed by an application. Of course, variations exist between applications. Some differences between applications may be characterized as differences in complexity. For example, Web applications are relatively unique in that, with respect to other applications, they have a more well restricted and encapsulated definition of their flow (as viewed from outside the application). A Web application's flow can be completely articulated by a series of requests for Uniform Resource Locators (URLs) and parameters passed on those requests that a client makes. Other applications may require more complex and encompassing representations to fully articulate the flow through the application. For example, application memory and register contents may need to be described in addition to the Application Program Interface (API) calls and parameters passed by the application.

Please replace paragraph [0029] with the following amended paragraph:

[0029] ~~FIGURE 12~~ FIGURE 12A and FIGURE 12B are [[is]] a flowchart illustrating one embodiment of request processing performed by an "error handling" function of the request manager.

Please add the following paragraph after paragraph [0029]:

[0029.1] FIGURE 13A illustrates an example of an error page displayed to the user in response to encountering an error while processing a request, according to one embodiment of the invention.

Please add the following paragraph after paragraph [0029.1]:

[0029.2] FIGURE 13B illustrates an example of an error page displayed to the user in the result of a retry, according to one embodiment of the invention.

Please add the following paragraph after paragraph [0029.2]:

[0029.3] FIGURE 14 illustrates an example of an error page for an error that is the result of a retry, according to one embodiment of the invention.

Please replace paragraph [0034] with the following amended paragraph:

[0034] The server 104 is shown as an application server for an application 108. The server [[1064]] 104 and the application 108 may reside on the same machine, or may themselves be distributed over the network 106. In general, the network 106 may be a local area network (LAN) and/or a wide area network (WAN). In a particular embodiment, the network 106 is the Internet, the server 104 includes a web server (e.g., an HTTP server) and the application is a Web application. In the case, the client 102

may include a web-based Graphical User Interface (GUI), which allows a user to display Hyper Text Markup Language (HTML) information. However, it is noted that aspects of the invention need not be implemented in a distributed environment, nor need the invention be Web-based. As such, a user may interact with the application on a local machine, such as is typically the case of a user using a word processor, for example.

Please replace paragraph [0040] with the following amended paragraph:

[0040] It follows, then, that a sister thread may not be instantiated once even one request is removed from the request queue 206. Given the overhead associated with maintaining a large queue, the request queue 206 will typically (but not necessarily) have a limited size and begin discarding requests according to a FIFO (first-in-first-out) model when the maximum size parameter is exceeded. This aspect is illustrated with respect to FIGURE 4 and FIGURE 5 which shows a request queue 206 having a defined maximum size of 1000. In the present illustration the request thread (RT) is handling request 1010 (counted from the first request to be placed on the queue). Since the queue size is 1000, the first 10 requests have been discarded (represented by the shaded area of the request queue 206 shown in FIGURE 5) and the earliest request remaining is request 11. Accordingly, while sister threads S1 and S2 continue to execute unaffected, no new sister threads may be created.

Please replace paragraphs [0042] – [0043] with the following amended paragraphs:

[0042] The foregoing assumes that a portion of the requests have been discarded from the request queue, thereby making it impossible to initiate new sister threads, regardless of how many sister threads are specified in the configuration file 204 of FIGURE 2. As a result, the user must elect to switch to one of the two existing sister threads and the earliest sister thread (S2 in the present example) is the farthest a user may go back in his/her request history. However, in the event all requests are preserved on the queue (since the beginning of the session), a user may choose to

begin executing from the beginning of the queue, regardless of whether any sister threads had been initiated.

[0043] In one embodiment, a user is given the option of either replaying or retrying execution from a selected sister thread. These functions are referred to herein as "replay processing" and "retry processing", respectively. "Replay processing" allows the user to step through the same execution path starting from the current position of the selected sister thread, which now becomes the request thread (RT' in the example illustrated in FIGURE 6). Thus, in the case of a "replay" the requests between when the error occurred and the selected sister thread are preserved and re-executed. In contrast, "retry processing" discards the requests between when the error occurred and the selected sister thread and allows the user to make different requests. In either case, the relevant portion of requests from the request queue 206 is represented by the shaded area 602 FIGURE 6.

Please replace paragraphs [0047] -- [0050] with the following amended paragraphs:

[0047] Referring first to FIGURE 7, an exemplary method 700 for processing a user's request is illustrated. In one embodiment, the method 700 is implemented by the process request function 302. Upon entering the method 700 (step 702), the request manager reads its configuration from the configuration file 204 (step 704) which may reside in some static storage, such as, a computer disk or database. Initially, the request manager goes to a wait state (step 706), waiting for an event of interest to occur. Instead of the request manager, a web server may handle implementation of this wait state. When an event of interest occurs during the wait state, the request manger examines the event to determine whether it is a termination event (e.g., a user session expiration, logging out, the user application ending, etc.) (step 708). If the event is a termination event, the request manager ends all requester threads 208 (step 710) and destroys the associated request queue 206 (step 712). When the request queue is destroyed, all of the user's associated information is cleared from the system. The system then, returns back to the wait state to wait for the next event of interest to occur.

If the event encountered is not a termination event, then the request manager examines the event to determine whether it is a user request (step 714). When the event is a user request, the request manager next determines whether the user is new or not (step 716). A user may be determined as new when no user specific information (i.e., user ID and associate state information) exists in the system for that user. If the user is new, then the request manager invokes the create services function 304 (step 720) for the new user, as described in more detail below with respect to FIGURE 8. The request manager then goes back to waiting for the next event. When the user is not new or after invoking the create services function 304 for new users, the request manager then services the request by invoking the service request function 306 (step 718), as described in more detail below with respect to FIGURE 9. Next, the request manager returns to the wait state until occurrence of the next event of interest.

[0048] As described previously, creating services may be invoked by the create services function 304. FIGURE 8 illustrates one embodiment of exemplary operations at step 720 for creating new services for a new user. Upon entering the method 720 (step 802), the request manager creates a new request queue for the new user (step 804). Then, the request manager creates an appropriate number of application threads for the user (step 806), as defined by the configuration file 204. Application threads may consist of a request thread 210 and the corresponding application state, and zero or more sister threads 212 (depending on the configuration) along with the corresponding application state. Next, the method 720 returns back to the process request function 302 (step 808).

[0049] As previously described, each new request may be serviced before being processed. FIGURE 9 illustrates one embodiment of exemplary operations at step 718 for servicing a new request. In one embodiment, the method 718 is implemented by the service request function 306. Upon entering the method 718 (step 905), the request manager creates a new request object for the request (step 910). The request object may contain all the relevant information and parameters required to process the user

request. By way of analogy, a request object, may resemble an HTTP request object. Next, the newly created request object is assigned a unique identifier (step 915).

[0050] In one embodiment, the unique identifier name may include information meaningful to the user based on the application flow or parameters. Alternatively, the request may be assigned a number. After being assigned a unique identifier, the request object is enqueued to the request queue 206 (step 920). The request, therefore, becomes the current request in the queue. Then, the request manager notifies the request thread 210 that a request is available (step 930) and waits for a response from the request thread 210 (step 940). After receiving a response form the request thread 210, the request manager determines whether the request was processed successfully (step 950). If the request thread 210 has encountered a failure in processing the request, the request manager invokes the error handler function 308 (step 990), as described in more detail below with respect to FIGURES 12A-12B ~~FIGURE 12~~. After results are returned from the error handler function 308, the request manager determines whether the error was handled by performing a replay or retry (step 960). When a replay has been performed, the request manager determines whether processing was successful (step 950). If so, the sister thread(s) 212 are notified (step 970). This notification signals the sister thread(s) 212 to get the next request from the request queue, as will be described in more detail below with respect to FIGURE 11. If the error was handled in any other way (e.g., by performing a retry, choosing to ignore and continue, or selecting optional debug actions) or after notifying the sister thread(s) 212, the method 718 returns back to the process request function 302 (step 995).

Please replace paragraphs [0053] – [0058] with the following amended paragraphs:

[0053] Returning to step 1014, if the request was not successfully processed (e.g., an error was encountered), the request thread discards any output prepared while processing the failed request (step 1020). The request manager 202 is notified of the failure (step 1022). In particular, the service request function 306 of the request

manager 202 is notified at step 940 of FIGURE 9. As will be described below with reference to FIGURES 12A-12B ~~FIGURE 12~~, the error handler 308 of the request manager 202 is responsible for presenting the user with the appropriate error page 203 (see FIGURE 2). The request thread then returns to a wait state (step 1002).

[0054] Referring now to FIGURE [[10]] 11, one embodiment of a sister thread wait loop 1100 is illustrated. The wait loop 1100 is implemented by each of the sister threads 212 and begins at step [[1002]] 1102, where the sister thread enters a wait state until receipt of an event. In the illustrative embodiment, the event is one of (i) a notification of successful performance of the current request (determined at step 1104), (ii) a notification to end processing (determined at step 1112), (iii) a notification to perform "retry processing" (determined at step 1118), (iv) a notification to perform "replay processing" (determined at step 1124) or a notification to perform "consume processing" (determined at step 1132).

[0055] Recall that a notification of successful performance of the current request is issued by the service request function at step 970 described above with respect to FIGURE 9. Receipt of such a notification indicates to the sister thread that it may get the next request from the request queue (step 1106). The sister thread then performs the request and updates its application state based on the type of request and actions undertaken as part of the request (step 1108). Any output generated for the user during performance of the request is discarded (step 1110), since any output to the user for this request will be presented at a different time by the request thread or the request manager. The sister thread then returns to a wait state (step 1102).

[0056] A notification to perform retry processing (determined at step 1118) is received from the error handler 308, as will be described in more detail below with respect to FIGURES 12A-12B ~~FIGURE 12~~. Upon receiving this notification the sister thread is logically named as the request thread (step 1120) and control flows to the request thread wait loop (step 1122), described above with respect to FIGURE 10.

Accordingly, the former sister thread performs retry processing as the request thread using new requests being input from the user.

[0057] A notification to perform replay processing (determined at step 1124) is received from the error handler 308, as will be described in more detail below with respect to FIGURES 12A-12B ~~FIGURE 12~~. Upon receiving this notification, the sister thread enters a loop (at step 1126) that is performed for each request in the request queue, except for the current request. In particular, the sister thread performs normal application processing and updates the application state accordingly (step 1128). Further, the sister thread discards any output generated for presentation to the user (step 1130). Once the sister thread has consumed all requests up to the current request, the sister thread is logically named the request thread (step 1120) and control flows to the request thread wait loop (step 1122), described above with respect FIGURE 10. In this way, all the requests (except the current request) are performed in a manner that is transparent to the user and the user is only presented with output that is appropriate to processing of the current request.

[0058] A notification to perform "consume processing" (determined at step 1132) is received from the error handler 308, as will be described in more detail below with respect to FIGURES 12A-12B ~~FIGURE 12~~. Upon receiving this notification, the sister thread enters a loop for each request in the request queue that the sister thread has been instructed to consume (step 1134). In particular, the sister thread performs normal application processing and updates the application state accordingly (step 1136). Further, the sister thread discards any output generated for presentation to the user (step 1138). Once the sister thread has consumed all appropriate requests, the sister thread returns to a wait state (step 1102).

Please replace paragraphs [0060] and [0061] with the following amended paragraphs:

[0060] Referring now to FIGURE [[12]] 12A and 12B, one embodiment of the operations performed at step 990 of FIGURE 9 is shown. As noted previously, these

operations may be performed by the error handler 308 of the request manager 202. Upon being invoked in response to unsuccessful processing of request, the error handler determines whether the error occurred while retrying a previous error (step 1202), i.e., when performing retry processing. If so, all debug functions are turned off and any tracing/debug information is stored in persistent storage for later use (step 1204).

[0061] If step 1202 is answered negatively, or after performing step 1204, the event handler determines whether any sister threads are available (step 1206). If not, the user may be presented with a generic error message (step 1208), i.e., one of the error pages 203 indicating the occurrence of an error, after which control may return to the service request function 306 (step 1210). If at least one sister thread does exist, the error handler may present one of the error pages 203 allowing the user to select from one or more options (step 1212). In particular, the error page may allow a user to select retry processing or replay processing. It is contemplated, however, that the particular options made available to the user will depend upon previous selections made by the user in response to errors. These aspects may be illustrated with respect to FIGURES 13-14. Illustratively, FIGURE 13 shows an error page 203A displayed to the user in response to encountering an error while processing request No. 32 (named "Run the current Query"). The user is given the option of performing retry processing, by selecting a first checkbox 1302, or replay processing by selecting a second checkbox 1304. Upon making a selection of either retry or replay processing, the user then selects the location from which retry/replay processing is performed. Each selectable location (represented by the checkboxes 1306) corresponds to a sister thread at that location. It is contemplated that, in the case of retry processing, the user may specify any request between the selected sister thread and the request thread which encountered the error. In this case, the selected sister thread performs consume processing up to the selected request (or the immediately proceeding request, depending on the particular implementation). In addition, the user is presented with debugging functions (checkboxes 1308) from which to select. Having made the desired selections, the user then clicks on the "Recover using these options" button 1310 to

perform the selected retry/replay processing. Alternatively, the user may choose to ignore the error and continue processing by clicking the button 1312.